



# SproutCore Render Delegates Specification

Version 1.2.1  
November 28, 2010

**Authors:**

Tom Dale [tdale@apple.com](mailto:tdale@apple.com)

**Changes:**

- Fixed an error in sample code on page 7.
- Removed in-depth discussion of themes. These will be covered in a separate specification.
- Changed update() method's elem parameter name to jquery.
- Added description of getChangedDisplayProperties().
- Reworded some sentences for clarity.

# Requirements

## Features

### Themeable Controls

SproutCore ships with many built-in controls, such as buttons, text fields, popup buttons, and so on. Often times, users want to theme these controls to match the desired look-and-feel of their application.

Currently, theming is done via custom CSS. However, users are bound by the DOM structure of the built-in controls. If they need more, fewer, or different elements, they must override the class and re-implement the render method. Render delegates should make it easy for views to offload their rendering to whatever their current theme is.

### Component Re-use

Some views are composites of the DOM of other views. For example, SC.ListItemView can optionally display a checkbox, icon, disclosure triangle, and label. These correspond to built-in controls, such as SC.CheckboxView and SC.DisclosureView. However, for performance reasons, we want to include the DOM representations of these views without instantiating full-blown views.

### Improved Performance

The SC.View API should nudge users in the direction of writing efficient custom views with fewer DOM operations.

## Constraints

### Backwards Compatible

Applications built using the SproutCore 1.4 gem should continue to work with the new system.

### Opt-in

Users of the framework should be able to implement custom views without having to create a separate render delegate object, unless they need the features listed above.

### Minimally Invasive

The implementation should require as few modifications to the existing view layer and controls as possible, while still fulfilling all requirements.

### Future-Proof

New API should be able to be adapted to up-and-coming HTML5 standards, such as SVG or canvas tags.

# Specification

## The Render Cycle

This section describes the render cycle as it exists today. Modifications to this implementation will be suggested in the following sections.

### Creating DOM

In order to display a view in the browser, SproutCore needs to create an HTML representation that can be appended to the DOM.

To do this, SproutCore calls the view's render method with two parameters: an SC.RenderContext instance, and a Boolean (`firstTime`) set to YES, indicating that no DOM representation exists.

When render is called with `firstTime` set to YES, it is the responsibility of the view to push strings of HTML into the SC.RenderContext. Once finished, those strings will be converted into an HTML element and appended to the DOM.

### Updating DOM

Certain properties of the view influence how it is displayed. For example, the appearance of SC.ButtonView will change when its `isEnabled` property changes. When one of these properties (called ***display properties***) changes, the render method will be called. This time, however, the `firstTime` flag will be set to NO. This is your cue that you should update the existing HTML representation instead of creating a new representation.

This is usually accomplished by retrieving a jQuery object for the HTML representation and making modifications to it, such as changing class names or changing the text content of the node.

### Problems

Both beginner and intermediate SproutCore users often write inefficient view code for two reasons:

1. Users are unclear what the `firstTime` flag means. Often, they will ignore it and push strings into the SC.RenderContext instance every time the render method is called. This is grossly inefficient because it causes the old HTML representation to be thrown out and a new one to be created and replaced in DOM.
2. Even users who are familiar with the `firstTime` flag and update their HTML representation using a jQuery object will often make more modifications than necessary. Often, they will make an update for every property, rather than just the property that changed.

## API Improvements

In order to clarify the difference between HTML generation and modification, the render method should be split in two.

### render

The render method will be used exclusively for HTML generation.

Its method signature looks like this:

```
render: function(context)
```

1. *context* — An instance of SC.RenderContext. The strings of HTML that represent your view should be pushed into this object.

Note that the `firstTime` parameter has been dropped; the render method is now called only when `firstTime` would be true in the previous implementation.

### update

A new method, `update`, will be called when one or more display properties are changed and an HTML representation already exists. In other words, it will be called where previously `render` would be called with `firstTime` set to `NO`.

Its method signature looks like this:

```
update: function(jquery)
```

1. *jquery* — A jQuery object corresponding to the HTML representation of the view.

## Detecting Display Property Changes

In order to efficiently update the appearance of a view, the `update` method must know what display properties have changed, and what their new values are.

SC.View adds a method called `getChangedDisplayProperties`. This method returns a hash that contains the set of values.

For example, imagine you had a view with the following display properties:

```
SC.View.create({  
  displayProperties: 'title value'.w()  
});
```

Now, after the initial HTML representation has been created, the user triggers an action that causes both the `title` and `value` properties of the view to be changed.

If you called `getChangedDisplayProperties()` on the view, you would receive a hash like the following:

```
{  
  title: 'Initial Title',  
  value: 'initial-value'  
}
```

Now imagine that only one of the display properties, value, is updated. Calling `getChangedDisplayProperties()` would now return:

```
{
  value: 'my-updated-value'
}
```

You can use the existence of a property in the hash to determine if it has changed or not. This is how the update method for this view might be implemented:

```
update: function(jquery) {
  var displayProperties = this.getChangedDisplayProperties();

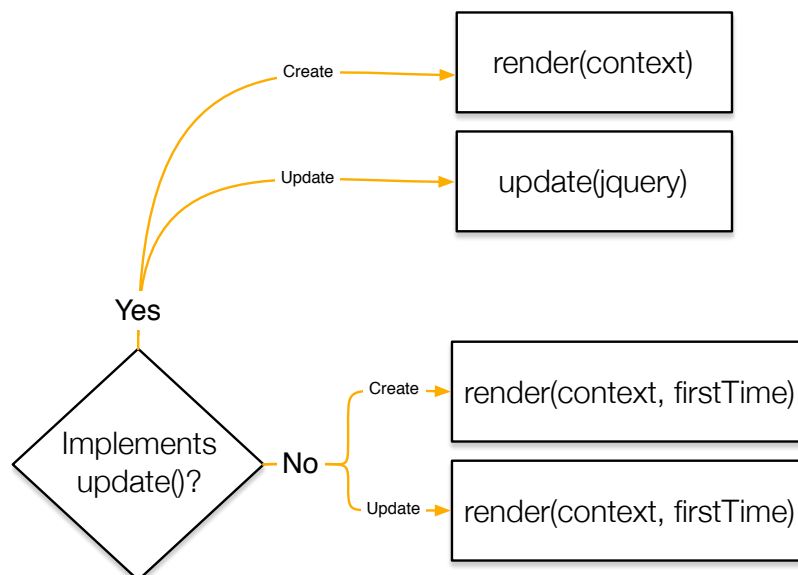
  if (displayProperties.contains('title')) {
    jquery.find('.title').text(displayProperties.title);
  }

  if (displayProperties.contains('value')) {
    jquery.find('.value').text(displayProperties.value);
  }
}
```

## API Versioning

To ensure backwards compatibility, older views that only implement the render method (and not update) will continue to behave the same way.

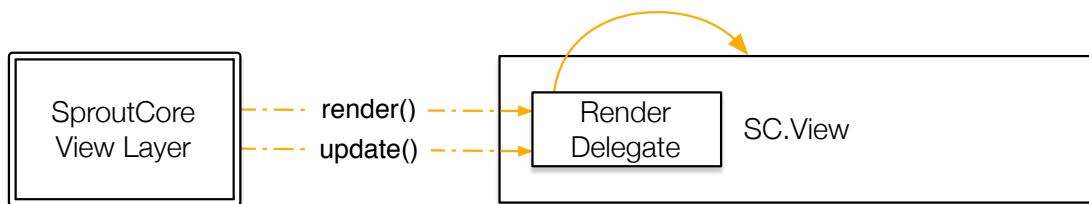
The view layer will use the existence of the update method to determine which version of the API should be used. For example, in a view that only has a render method, it will continue to get called for both creation and update, with the `firstTime` parameter set to the appropriate value.



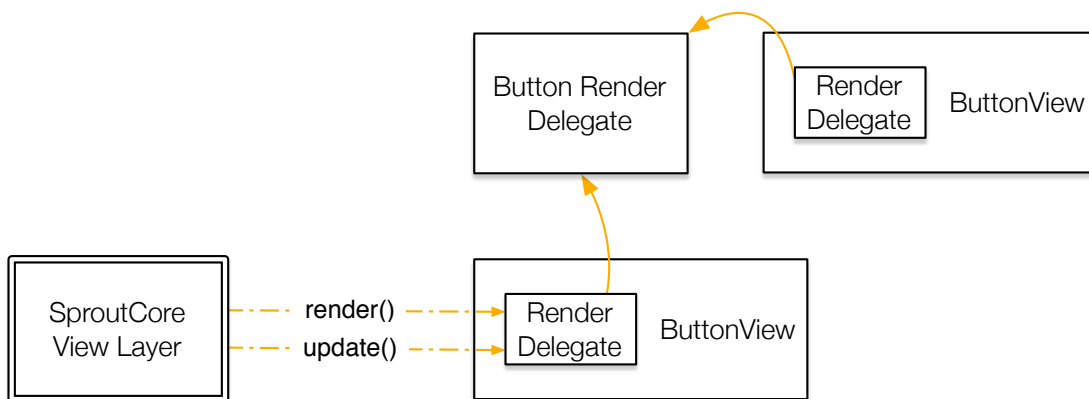
## Render Delegates

Any object that implements render and update is considered to implement the **render delegate protocol**. Any object that implements the render delegate protocol can be used as a **render delegate**.

By default, an instance of SC.View is treated as its own render delegate. When the SproutCore view layer calls render or update, it will be called on the view because its renderDelegate property is set to null.



In some cases, you may want the logic for maintaining the HTML representation outside of the view. You can set the render delegate to be any object that implements the render delegate protocol. Because render delegates are stateless, you can share the same instance among multiple views.



If the render delegate is not a view, the first parameter passed to the render and update methods is a reference to the view being represented. Render delegates should query this object to determine the state of the view to render.

```
render: function(dataSource, context)
```

1. **dataSource** — The view that contains the display properties that determine how the view should be rendered.

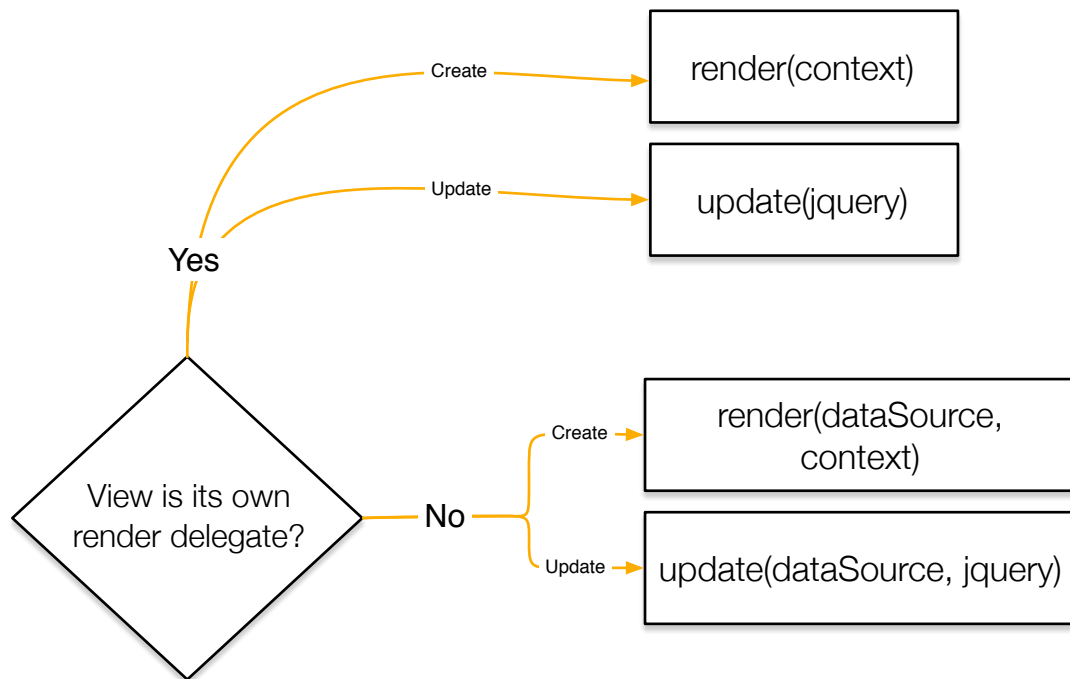
2. *context* — An instance of `SC.RenderContext`. The strings of HTML that represent your view should be pushed into this object.

**update:** `function(dataSource, jquery)`

1. *dataSource* — The view that contains the display properties that determine how the view should be rendered.
2. *jquery* — A jQuery object corresponding to the HTML representation of the view.

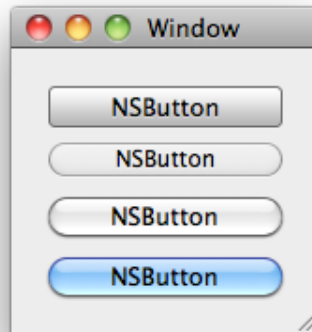
For example:

```
render: function(dataSource, context) {  
  context.setClass('sel', dataSource.get('isSelected'));  
}
```



## Render Delegates & Themes

Controls can take on a different appearance depending on their context. For example, in Mac OS X, consider these four variations of `NSButton`:



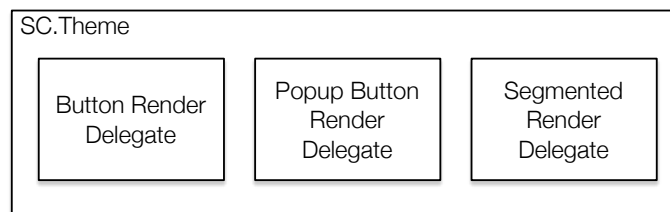
They are all instances of NSButton and exhibit identical behavior. The only difference is their visual appearance.

In SproutCore, a control's theme dictates how it is rendered. Themeable controls retrieve their respective render delegate from an instance of SC.Theme and let it be responsible for creating the HTML representation.

To learn more about themes, see the specification at <http://www.sproutcore.com/specifications/>.

## Registering a Render Delegate

The collection of render delegates that compose a unified appearance for controls is contained in an instance of SC.Theme.



Adding a render delegate to a theme is as simple as adding a property to it:

```
MyCustomTheme.buttonRenderDelegate = SC.Object.create({
  render: function(dataSource, context) {
    // Your render code here.
  },

  update: function(dataSource, context) {
    // Your update code here.
  }
});
```



## Building Themeable Controls

You can tell a view to retrieve a render delegate from the theme by setting its `renderDelegateName` property. The view will use this property to find the associated render delegate from the current `SC.Theme` object.

For example, `SC.ButtonView` has its `renderDelegateName` property set to `"buttonRenderDelegate"`. When a new instance of `SC.ButtonView` is created, it looks up the `buttonRenderDelegate` property on its theme and saves that object to the `renderDelegate` property.

